On cherche dans ce problème à modéliser une structure de dictionnaire : à un mot, on souhaite associer une définition. Afin de simplifier l'étude du modèle, les définitions seront représentées par des entiers naturels. Trouver la définition d'un mot reviendra donc à trouver son numéro associé. Dans tout le sujet, un mot sera représenté par une liste de lettres :

```
type mot = char list;;
```

Les opérations auxquelles on s'intéressera pour cette structure sont :

- Recherche de définition.
- Ajout d'une association (mot, définition) à un dictionnaire.
- Suppression d'une association (mot, définition).
- Fusion de deux dictionnaires.

Dans tout le sujet, sauf précision contraire, on écrira des fonctions récursives sans utilisation de références.

I Représentation par des listes

On propose dans un premier temps de représenter un dictionnaire par une liste de mots et de leurs définition :

```
type dico = (mot * int) list;;
```

Question 1. Expliquer succinctement quelle opération est la moins coûteuse en temps avec cette structure de dictionnaire.

Question 2. Écrire une fonction `plus_long : dico -> mot` qui renvoie le mot le plus long dans un dictionnaire. On renverra un mot vide si le dictionnaire est vide. La fonction ne devra pas calculer la taille d'un mot plus d'une fois.

Question 3. Écrire une fonction `prefixe mot -> mot -> bool` qui teste si un mot est préfixe d'un autre.
u est préfixe de v si v peut s'écrire uw où w est un mot.

Question 4. Écrire une fonction `egaux : mot -> mot -> bool` qui teste si deux mots sont identiques.

Question 5. En déduire une fonction `recherche : mot -> dico -> int` qui renvoie le numéro associé à un mot. On renverra -1 si le mot n'est pas présent dans le dictionnaire.

Question 6. Prouver la terminaison de la fonction précédente.

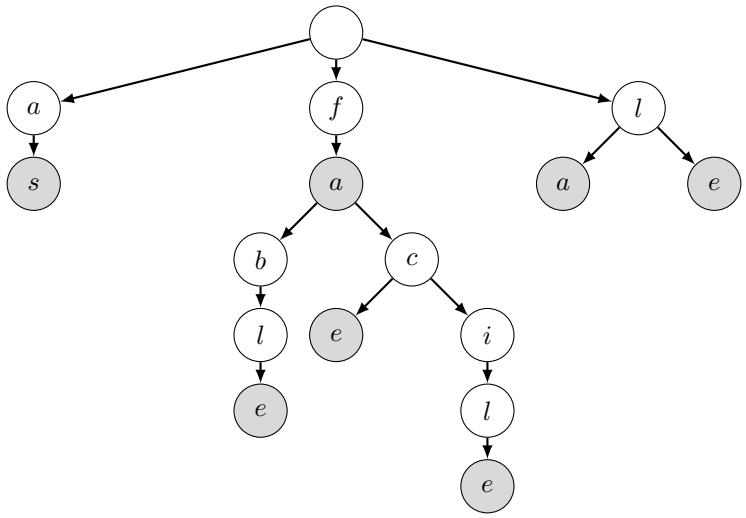
Question 7. En notant n le nombre de mots dans le dictionnaire et k la taille du mot cherché, déterminer la complexité temporelle dans le pire des cas de la fonction `recherche`. On explicitera un cas permettant d'atteindre cette complexité.

Question 8. Écrire une fonction `suppression : mot -> dico -> dico` qui prend en argument un mot et un dictionnaire et renvoie le dictionnaire où on a supprimé le mot et sa définition s'il est présent. On supposera que le dictionnaire ne contient pas de doublon.

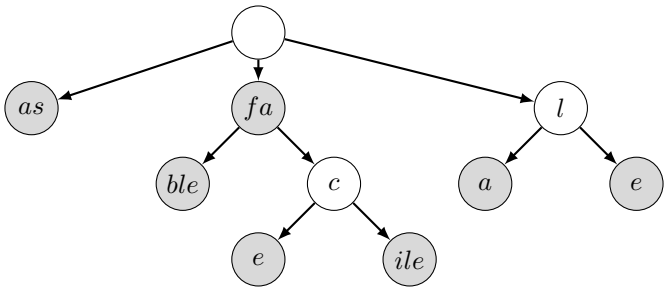
II Arbres lexicographiques et Patricia

Un arbre lexicographique est un arbre (pas nécessairement binaire) pour représenter des dictionnaires. Les nœuds autres que la racine sont étiquetés par des lettres. Le mot associé à un nœud correspond à la suite des lettres lues pour arriver à ce nœud. Un nœud peut être terminal (le mot associé au nœud est dans le dictionnaire) ou non-terminal (le mot associé au nœud n'est pas dans le dictionnaire).

Par exemple on représente un arbre lexicographique associé au dictionnaire contenant les mots as, fa, $fable$, $face$, $facile$, la et le par :



Un arbre Patricia (pour *Practical Algorithm To Retrieve Information Coded In Alphanumeric*) fonctionne sur le même principe qu'un arbre lexicographique, mais diminue le nombre de nœuds en étiquetant par des mots plutôt que par les lettres. Par exemple, l'arbre Patricia associé à l'arbre lexicographique précédent est :



On représentera un arbre Patricia par le type suivant :

```
type patricia = N of mot * int * patricia list;;
```

L'entier indique ici si le nœud est non-terminal (il vaut alors -1), ou terminal (il vaut alors le numéro de la définition associée au mot, entier naturel).

Par exemple, l'arbre associé à l'exemple précédent est (avec des numéros arbitraires) :

```
N([], -1, [ N(['a';'s'], 1, []) ;
            N(['f';'a'], 2, [ N(['b';'l';'e'], 3, []) ;
                              N(['c'], -1, [ N(['e'], 4, []) ;
                                             N(['i';'l';'e'], 5, []) ]) ]) ;
            N(['l'], -1, [ N(['a'], 6, []) ;
                           N(['e'], 7, []) ]) ]);;
```

Les règles de construction d'un arbre Patricia sont les suivantes :

- La racine n'est pas terminale.
- Toutes les feuilles (sauf éventuellement la racine) sont terminales.
- Les mots des fils d'un nœuds sont toutes distinctes et par ordre strictement croissant de leurs premières lettres.

Question 9. Écrire une fonction `taille : patricia -> int` qui compte le nombre de mots présents dans un dictionnaire représenté par un arbre Patricia.

Question 10. Écrire une fonction `prefixage : mot -> dico -> dico` qui rajoute un préfixe à tous les mots d'un dictionnaire (au sens de la première partie). Par exemple, un appel à `prefixage` avec fa sur $[ble; ce; cile]$ renverra $[fable; face; facile]$. On autorisera l'opérateur `@` de concaténation.

Question 11. En déduire une fonction `dico_patricia : patricia -> dico` qui renvoie un dictionnaire à partir d'un arbre Patricia. On autorisera l'opérateur `@` de concaténation.

Question 12. Écrire une fonction `compare : mot -> mot -> bool` qui teste si un la première lettre d'un mot est strictement inférieure à la première lettre d'un autre mot. Par exemple, $as < fa$, mais $fable \not< facile$.

Question 13. En déduire une fonction `est_patricia : patricia -> bool` qui vérifie si un arbre est bien un arbre patricia.

Question 14. Écrire une fonction `recherche : mot -> patricia -> int` qui renvoie le numéro associé à la définition d'un mot donné en argument dans un arbre Patricia. On renverra -1 si le mot n'est pas trouvé.

Question 15. Déterminer la complexité temporelle de la fonction précédente.

Question 16. Écrire une fonction `creation : mot * int -> patricia` qui crée un arbre Patricia ne contenant qu'un seul mot et sa définition, donnés en argument de la fonction.

Question 17. Écrire une fonction `ajout : mot * int -> patricia -> patricia` qui ajoute un mot à un arbre Patricia. On commencera par détailler à la main les étapes de l'ajout des mots $facteur$, *lampe* et *lame* dans l'arbre.

Question 18. En déduire une fonction `patricia_dico : dico -> patricia` qui construit un arbre Patricia à partir d'un dictionnaire.

Question 19. Écrire une fonction `suppression : mot -> patricia -> patricia` qui supprime une association dans un arbre Patricia.

Question 20. Écrire une fonction `fusion : patricia -> patricia -> patricia` qui fusionne deux arbres Patricia.